Fight Club Planner

Analysis and Design Document

Student: Stancu Gabriel - Iulian

**Group: 30432**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 6

4. UML Sequence Diagrams 9

5. Class Design 9

6. Data Model 12

7. System Testing 13

8. Bibliography 13

1. Requirements Analysis

# Assignment Specification

Design and implement an application that helps MMA tournaments manage their scheduled fights better while ensuring covid safety standards.

# Functional Requirements

A manager should be able to create a tournament and invite fighters to sign up for a venue. Each tournament requires weekly or monthly matches to generate traction and revenue within the tournament period.

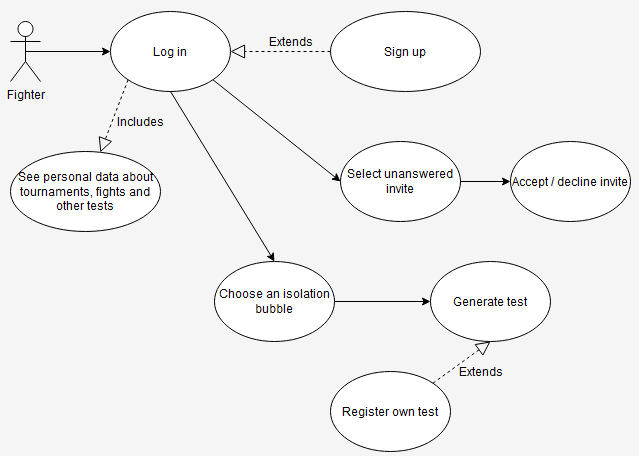
After they sign up, in order to ensure proper safety standards, fighters are required to present themselves at a tournament isolation bubble with a test which will be recorded and they will immediately be tested again on site. If the “arrival” test is positive the fighter is then moved to quarantine until the test results are negative again.

In order for a fighter to take part in a tournament they require at least 3 weeks of negative test history after which they can be matched up with similar fighters of their caliber.

A manager should be able to see in real time the tournament schedule being populated with eligible fighters (at least bi-weekly).

# Non-functional Requirements

* Implement and test the application
* Commit the work you do on your Git repository. Do it iteratively as you progress, not all at once (this will incur a penalty on your final mark)
* Use any OOP language you like. Non-exhaustive: Python, C#, Java, Ruby, C/C++, JS+Typescript
* Use a CQRS architecture, use a mediator pattern to handle requests
* Use a decorator pattern for changing the color of the fighters’ status (green for safe, red if they have a positive test within 3 weeks and grey otherwise)
* The data will be stored in a database
* All the inputs of the application will be validated against invalid data before submitting
* the data and saving it in the database.

2. Use-Case Model

**Use case**: Fighter accepts tournament invites

**Level**: user – goal

**Primary actor**: Fighter

**Main success scenario**: the fighter logs in, checks the invites table, clicks on a non – answered one, accepts / declines it

**Extensions**: the fighter is not registered, so he has to sign up first

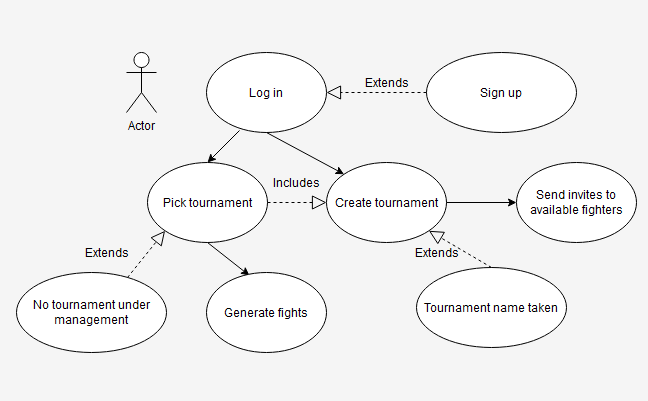
**Use case**: Fighter tests himself

**Level**: sub-function

**Primary actor**: Fighter

**Main success scenario**: the fighter logs in and tests himself for being eligible for fights. In order to achieve this, a fighter should test himself regularly (at least one test per week and all tests negative) for 3 consecutive weeks.

**Extensions**: the fighter is not registered, so he has to sign up first and register a test taken by himself before being tested

****

**Use case**: Manager creates tournaments and invites fighters to it

**Level**: user – goal

**Primary actor**: Manager

**Main success scenario**: the manager logs in, creates a new tournament, then sends invites to available fighters

**Extensions**:

* the manager is not registered, so he has to sign up first.
* the tournament name is taken, so he needs to rename it.

**Use case**: Manager generates fights and sets the winner

**Level**: sub-function

**Primary actor**: Manager

**Main success scenario**: the manager picks a tournament he is responsible for, selects a date for starting the fight week (by default it is the current date) and the system generates fights between the available fighters; after fight has taken place, the manager can pick the winner. The use case assumes the manager will wait until a specified moment to generate the fights, so that the fighters can test themselves and become eligible. If a fighter is found positive or he is not tested regularly, he won’t be considered eligible for fighting.

**Extensions**: the manager has no tournament under management, so he has to create one and invite fighters to it

3. System Architectural Design

**3.1 Architectural Pattern Description**

Because the application needed to be developed under the CQRS architecture, several changes needed to be performed from the 2nd assignment. The client now sends commands to the server (instead of DTOs), which are intercepted by controllers, but this time they are not handled by services, but by a common mediator, which redirects the command to an event handler. The event handler is chosen dynamically based on the type of the command, using the required repositories for achieving the final goal of the request, just like the services did in the previous version. The improvement this change brings is the fact that the controllers are less tightly – coupled, having only a dependency on the common mediator, instead of using several services references for handling the incoming requests.

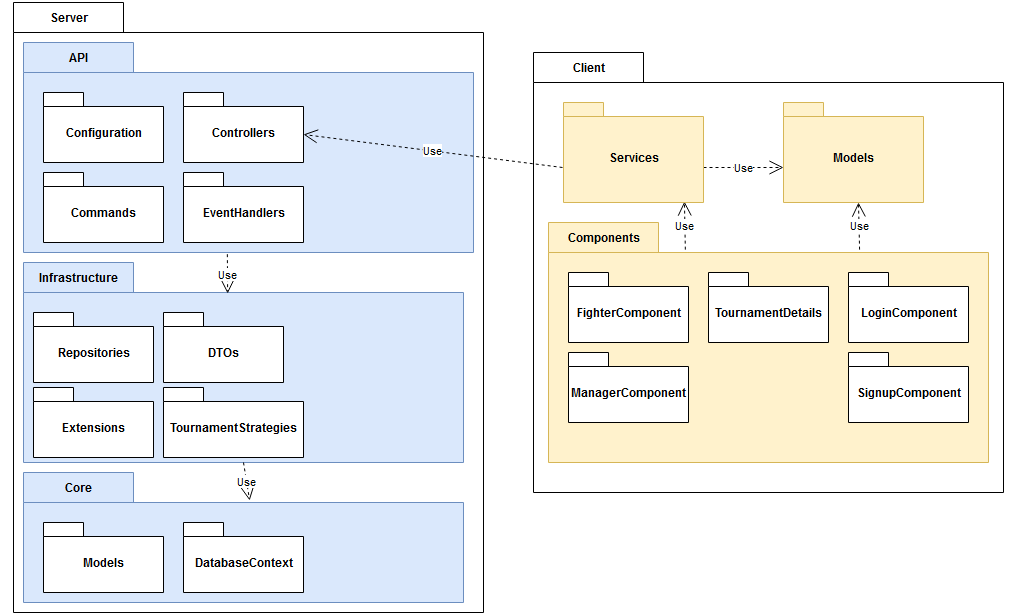
The client was developed in a more component-based manner, organizing the code, classes and files by the final functionality they serve. We have separate folders for models and services, while the components containing display and business logic functionalities are grouped by their final purpose. In other words, any view that needs to be rendered in the browser’s page, has the template, styles and backing functionality grouped in its own folder. This structure was encouraged by the use of the Angular framework, which is natively built for being used this way. The client application is a single page application, the final rendering of the content being a switch between the components that need to be rendered, depending on the actions the user takes while using the application. For communicating with the server side, services were implemented, having a mapping of 1 on 1 with the controllers from the server. In other words, each view has its own service which communicates with the controller responsible for handling the requests available in that view. This helps for a better organization of the code and allows easier improvements over time (for future updates). The services form the requests with the required data, send them to the server, waiting (asynchronously) for the reply from the server. The client has no direct access to internal layers of the server (for instance the business logic or data layer) or other application resources (for instance the database). The only way of communicating with the server is through requests that are sent to the controllers of the server only.

The main architectural pattern used for developing the server is the layered pattern. We divided the application in 4 distinct layers:

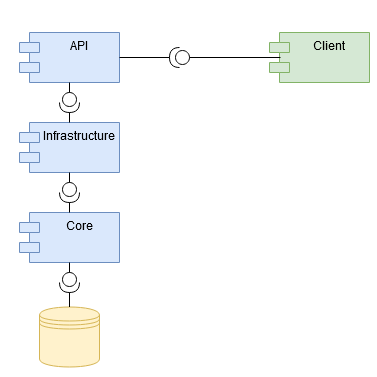
* The data layer [Core]: the layer responsible for mapping the database entities into program classes; the models are 1 on 1 mappings of the database entities. For better mappings we used the EF Core annotations. This layer also contains the database access point, the database context, also supported by the EF Core framework.
* The business logic layer [Infrastructure]: the layer responsible for the functionality of our application. This is a rich layer which will be divided in more discrete layers in future updates, which contains several class groups of functionalities: repositories, DTOs, extensions of certain classes used by upper layers, strategies for different features (such as the tournament creation feature). The repositories act as DAOs, providing methods that reflect queries on the database. These repository methods are mostly accessed by the services, which work with the results of these methods, together with a data mapper (AutoMapper for this project) that shapes the data in a flatter way (the format described by the associated DTOs and commands from the upper layer).
* The presentation layer: it contains the controllers of the server (the hooks reachable by HTTP requests received from clients), the mediator, the command types event handlers and the configuration classes. Such configuration classes are Startup (where we configure the dependency injection container with services required in lower layers, or where we configure the pipeline of a request by setting the middleware components, or the database connection string) or the Program class (the entry point in the server application). The controllers provide points of access for requests in various formats, by providing an URL for the request, the expected data and the format of the response that will be sent to the clients. The following request types are supported: GET, POST, PUT. The commands are objects that encapsulate the data sent from the client and their requirement under the form of an event. The mediator is a common entry that intercepts these events and redirects them to the handler capable of solving the requests. The handlers have references to the needed repositories for solving these requests. Finally, after the retrieved data is shaped / mapped, the handlers make sure the results are sent back up to the mediator, which then returns the final result to the controller, which will then send the final object(s) back to the client.

**3.2 Diagrams**

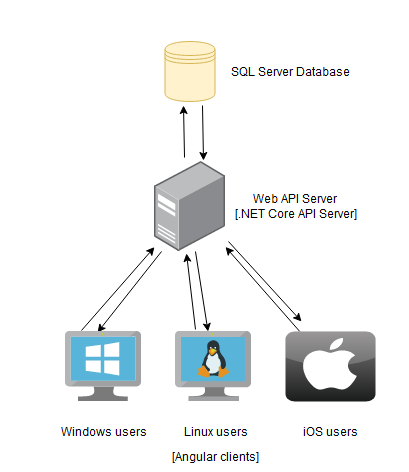
**Package diagram:**

****

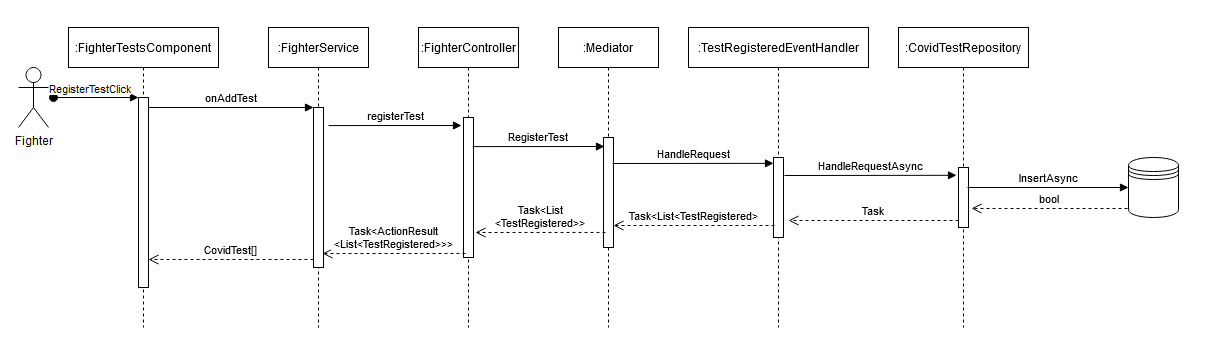
**Component diagram:**

****

**Deployment diagram:**

****

4. UML Sequence Diagrams



5. Class Design

**5.1 Design Patterns Description**

Because the tournament matches generation policy might change over time, we needed a behavioral design pattern to handle this dynamic change. For this purpose, we used the Strategy design pattern. At the core of this pattern we have the IMatchStrategy interface, which provides a signature method for creating a new fight to be appended to the list of fights of the given tournament. It also provides a default method for getting the last date of a match in a tournament, to leave a required gap after it, or the date for the first fight (the maximum between the start date of the tournament or the current date).

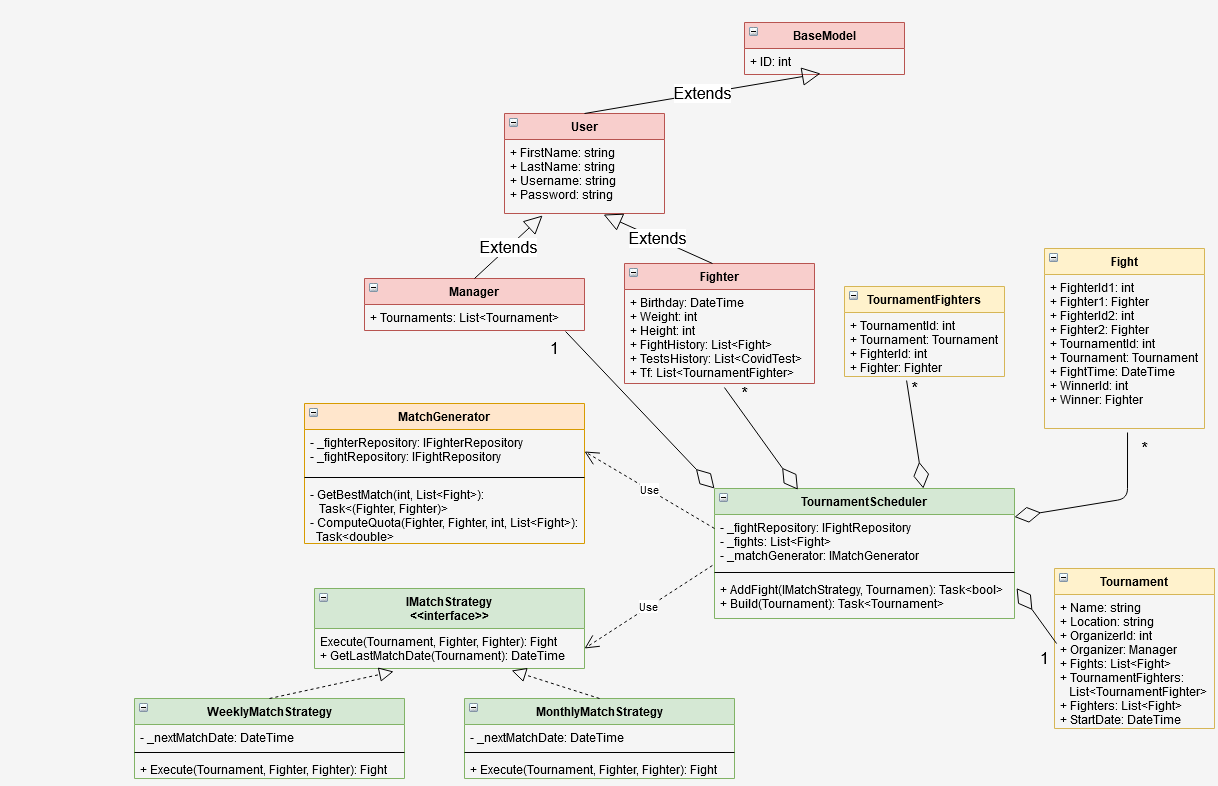
The interface is implemented by the two strategies we have in our project: MonthlyStrategy (which generates two matches per month) and WeeklyStrategy (two matches per week) by implementing the previous interface and giving a concrete implementation for the generation functionality.

Finally, the organizer of this whole pattern is the TournamentScheduler class. After it called the addition of the required number of fights, its Build() method is called for inserting the generated entity in the database (through the database context) and returning to the caller the object representing this entity.

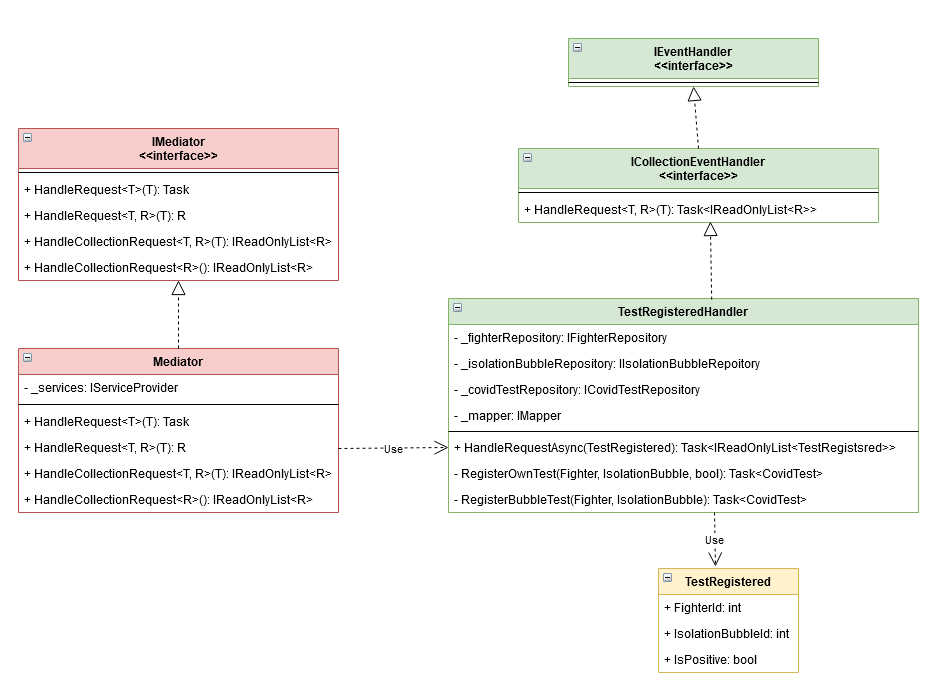
Because we are required to implement CQRS architecture and employ the Mediator design pattern, we provided the following solution: instead of using DTOs (only) for sending requests from the client to the server, we use Commands, which are intercepted by the Mediator, which redirects the task of handling the command to one of the handlers it manages. The handlers are of 4 types: ICollectionEventHandler (receives a parameter and returns a collection), IEntityEventHandler (receives a parameter and returns a single entity), IParameterlessCollectionEventHandler(receives no parameter and returns a collection) and IVoidEventHandler (receives a parameter and returns nothing). For each type of command there is a concrete event handler implementing one of the 4 interfaces.

For displaying dynamic behavior at runtime, we implemented the Decorator design pattern. It is used for displaying the status of a fighter in the tournament information page, for displaying the invite status and test result in the fighter window. It takes the base entity (received from server) and enhances it with a method that will be implemented by the whole tree: the concrete component will extend an abstract component, that is extended by an abstract decorator, which is turn is extended by concrete decorators, depending on the behavior we want to showcase. In our case, we color a table entry depending on the status of the base component in the tree: we color a fighter in gray if he has no tests in the previous 3 weeks, green if he is eligible, red if sick. The same applies for tests: green if negative, red if positive, for fights: green if won, red if lost, green otherwise and also for invites: green if accepted, red if declined, not colored if not answered yet.

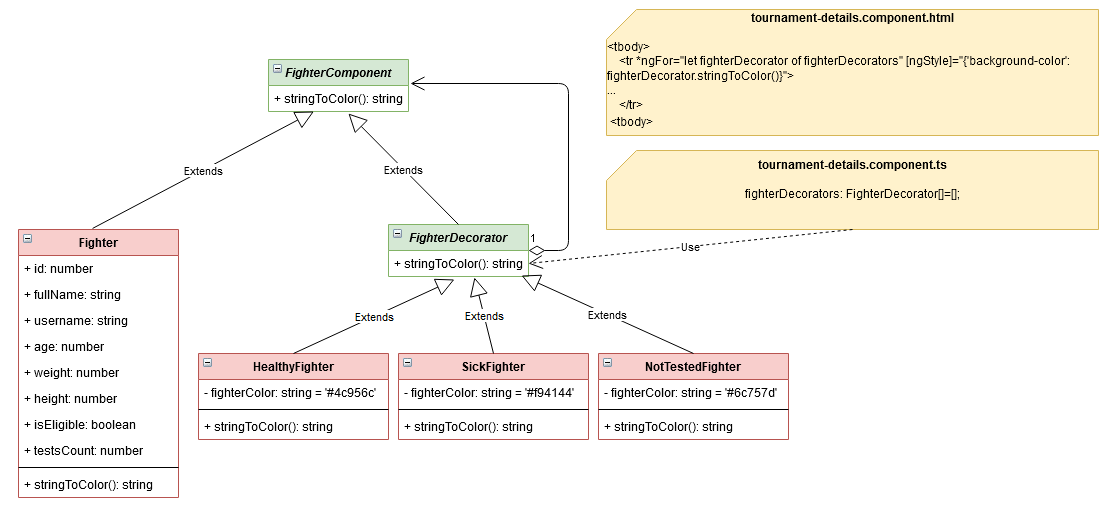
**5.2 UML Class Diagram**

**

On the same diagram the previously mentioned variation of the Player – Role design pattern can be noticed. Although the Role class misses, the principle is the same: we have a common ancestor: the User class, that would have a dependency on the Role class, if we implemented it, and then the sub – roles of Manager and Fighter which a user can take. However, because the roles cannot change over time, the Role dependency would become redundant and we can use the type of the class at runtime to determine the role of a certain User instance.

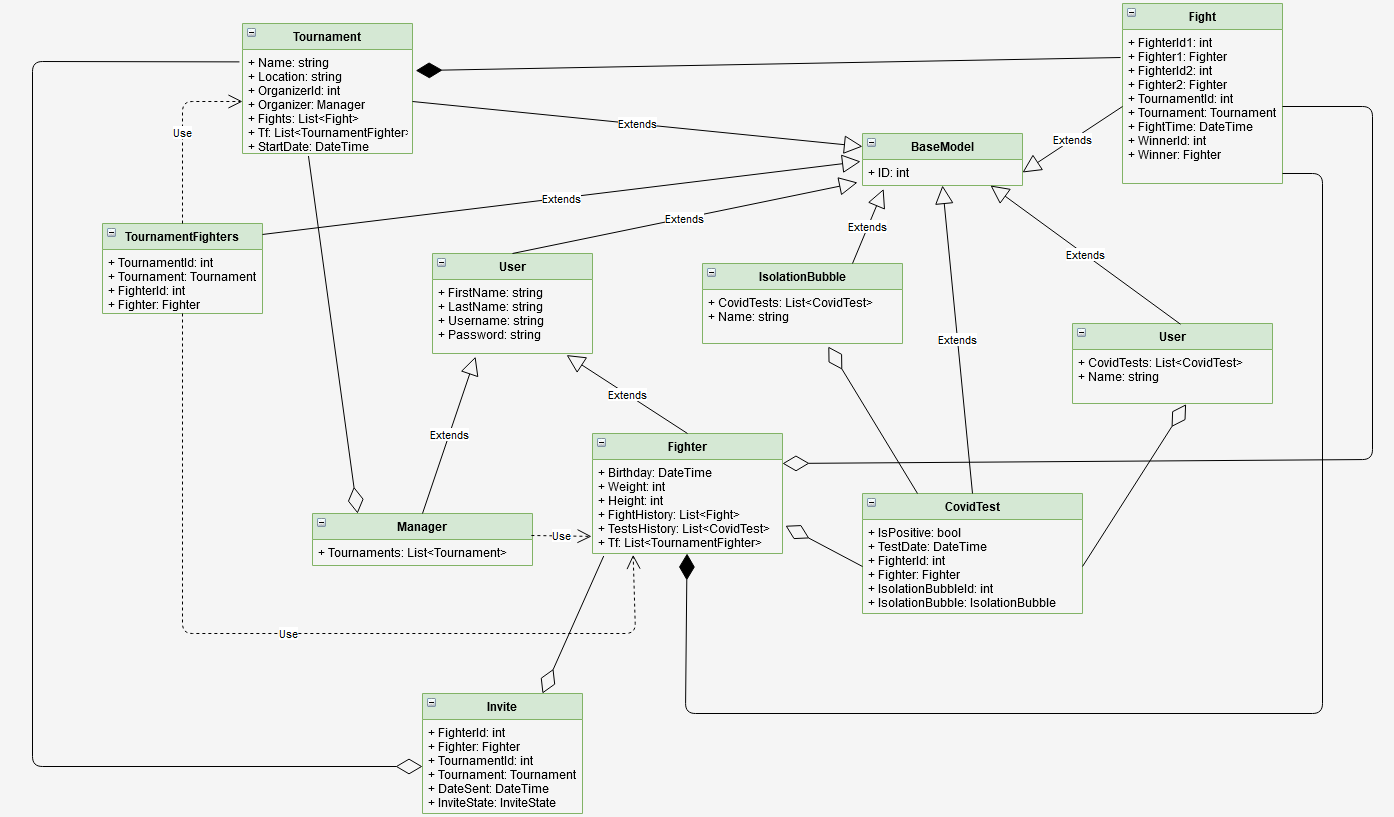


On the previous diagram, the Mediator design pattern can be observed. The command sent from the client is the TestRegistered object. It is intercepted by the Mediator and redirected to the TestRegisteredHandler for handling it, instead of using a service, as we did in the 2nd assignment. On the diagram we only displayed one case, but all types of commands are handled the same way, through the mediator, so a full migration was performed from the previous version of the application.



On the previous diagram, the concrete component is the Fighter class. It extends the abstract component (FighterComponent), which is also extended by the abstract decorator (FighterDecorator). Finally, the abstract decorator is extended by the 3 concrete decorators, that add dynamic functionality to the final object that is displayed: depending on the status of the fighter, they return different color hexadecimal codes, that will be used by the ngStyle directive to render the final color. Instead of having a table / list of fighters, however, we have a list / table of FighterDecorators. The same principle was applied for invites, fights and tests in the fighter window.

6. Data Model

**

The data layer in our application, the Models package more precisely, mirrors almost perfectly the data shape we have in the database. However, some additional properties are created and not mapped in the database (the [NotMapped] annotation in EF Core). These properties are used for easier computation of certain information (if a fighter is eligible for taking part in tournaments, for instance), but they are used in the application, not from the data’s point of view. In the diagram above we removed these properties and left only the data properties that reflect the database entities and their relationships (foreign keys, dependencies etc.).

7. System Testing

For testing we proposed a small number of tests, due to time limitations on the project delivery. However, the critical parts of the project were tested: the database connection, the builder design pattern functionality and the eligibility of a fighter for taking part in tournaments. We are also testing the fighter controller. In future versions of the app, the testing project will be refactored and it will test all controllers. The unit testing approach was unit testing, using NUnit testing.

* Database connection: we established a database connection and we checked against a select command (retrieve an instance we know is in the database from the setup) and later delete it. A cleanup method was provided to ensure no testing entities remain in the database.
* For testing the eligibility of a fighter, we test 3 cases:

1. The fighter is eligible (has no positive test and has tested regularly over the last 3 weeks).
2. The fighter is not eligible because a test was found positive.
3. The fighter is not eligible because he was not tested regularly over the last 3 weeks (at least one test each week).

* For the fighter controller, we are testing the requests it can receive:

1. Answering an invite
2. Registering a test
3. Getting all the details of a fighter
4. Getting the isolation bubbles where a fighter can test himself

Future releases of the application will contain unit testing for larger areas of the project, including the services (the helpers from the business layer), other database accessors etc.

8. Bibliography

1. [**https://www.w3schools.com/css/**](https://www.w3schools.com/css/)

**2.** [**https://www.dofactory.com/net/design-patterns**](https://www.dofactory.com/net/design-patterns)